

Parallel Hybridization for SAT: An Efficient Combination of Search Space Splitting and Portfolio

Rodrigue Konan Tchinda, Clémentin Tayou Djamegni

▶ **To cite this version:**

Rodrigue Konan Tchinda, Clémentin Tayou Djamegni. Parallel Hybridization for SAT: An Efficient Combination of Search Space Splitting and Portfolio. Revue Africaine de Recherche en Informatique et Mathématiques Appliquées, 2021, Volume 34 - 2020 - Special Issue CARI 2020, 10.46298/arima.6750 . hal-02926523v2

HAL Id: hal-02926523
https://hal.science/hal-02926523v2

Submitted on 17 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

ARIMA

épisciences

Parallel Hybridization for SAT: An Efficient Combination of Search Space Splitting and Portfolio

Rodrigue Konan Tchinda[*1] **and Clémentin Tayou Djamegni**[1]

[1]University of Dschang, Cameroon

*E-mail : rodriguekonanktr@gmail.com

Submitted on September 1, 2020 - Published on January 15, 2021
Volume : **34** - Year : **2020**
Special Issue : **Volume 34 - 2020 - Special Issue CARI 2020**

Abstract

Search space splitting and portfolio are the two main approaches used in parallel SAT solving. Each of them has its strengths but also, its weaknesses. Decomposition in search space splitting can help improve speedup on satisfiable instances while competition in portfolio increases robustness. Many parallel hybrid approaches have been proposed in the literature but most of them still cope with load balancing issues that are the cause of a non-negligible overhead. In this paper, we describe a new parallel hybridization scheme based on both search space splitting and portfolio that does not require the use of load balancing mechanisms (such as dynamic work stealing).

Keywords

SAT; portfolio; search space splitting; parallel hybridization

I INTRODUCTION

The Boolean Satisfiability Problem (SAT) consists of determining whether there exists an assignment of truth values to variables of a given propositional logic formula in order to make it evaluate to true. This problem of great importance in Computer Science is a subject of special attention since the advent of modern SAT solvers based on the so-called CDCL (Conflict-Driven Clause Learning) procedure [4, 7, 8, 10, 13]. SAT is known to be NP-complete [1] and therefore is very hard to solve (unless $P = NP$). Despite this theoretical hardness, recent researches in the last two decades have resulted in very efficient SAT solvers that are able to solve industrial formulas with millions of variables and clauses in very little time. This great success has led to their use in many other fields including formal verification, planning, bioinformatics, cryptanalysis, etc. Faced with ever increasing need of performance and because of microprocessors' frequency limitation due to technological constraints, the efficiency of current state-of-the-art sequential SAT solvers is no longer sufficient since many industrial instances are still out of their reach.

Researchers then turned to efficient parallelization of SAT [17, 24, 26, 31] since the increase in the computing power of microprocessors today has resulted in an increase in their number of cores. Nowadays, there are two main approaches used for this purpose namely search space splitting and portfolio, each of which having its strengths and weaknesses.

Many hybrid approaches [15, 21, 22, 26, 27] have been proposed for parallel SAT solving but most of them still suffer from load balancing issues which are the cause of a non-negligible overhead. Our aim in this paper is to propose a new hybridization scheme that overcomes workload balancing issues while inheriting the best features of search space splitting and portfolio approaches.

The remainder of this paper is organized as follows: Section II briefly recalls some basic concepts and gives an overview of parallel SAT solving. Section III exposes our hybrid approach and in Section IV we present our implementation followed by experimental results. We present some related work in Section V and finally conclude our paper in Section VI while pointing out some future research directions.

II PRELIMINARIES

We assume that the reader is familiar with the Boolean Satisfiability problem; however, we recall here some basic concepts used in solving this problem. The interested reader may refer to [20, 32, 33] for more details.

2.1 Definitions and Notations

A *Boolean variable* is a variable that can be assigned only two possible values: *true* ($\top$ or 1) or *false* ($\bot$ or 0). A *literal* is either a Boolean variable (*positive literal*) or its negation (*negative literal*). A *clause* is a disjunction of literals (i.e. literals connected with $\vee$). Propositional formulas are commonly represented in *Conjunctive Normal Form (CNF)* i.e. as a conjunction of clauses (clauses connected with $\wedge$). A CNF formula can be seen as a set of clauses where each clause is a set of literals. An *interpretation* or *assignment* (or a *truth assignment*) is a map $\sigma : \mathcal{V} \rightarrow \{0, 1\}$ which associates a truth value to each variable of $\mathcal{V}$. If $\mathcal{V}$ is a subset of variables of a formula $\mathcal{F}$ then σ is called a *partial assignment* of $\mathcal{F}$. A truth assignment $\sigma : \mathcal{V} \rightarrow \{0, 1\}$ can be represented as a set of literals $\mathcal{I}$ such that for every variable $x \in \mathcal{V}$, $x \in \mathcal{I}$ iff $\sigma(x) = 1$, $\neg x \in \mathcal{I}$ iff $\sigma(x) = 0$ and x is unassigned iff $\{\neg x, x\} \cap \mathcal{I} = \emptyset$. A literal l is satisfied under an assignment $\mathcal{I}$ if $l \in \mathcal{I}$ and falsified if $\neg l \in \mathcal{I}$ where $\neg l$ denotes the opposite literal of l i.e. the literal that evaluates to *true* when l is *false* and *false* when l is *true*. A clause is said to be *satisfied* under an assignment when it contains at least one satisfied literal, and is *falsified* if all its literals are falsified. An empty clause is a clause with no literals: it is always falsified. A clause is *unit* under a partial assignment when all its literals are falsified except one which is unassigned. A CNF formula is satisfied under an interpretation $\mathcal{I}$ if all its clauses are satisfied under $\mathcal{I}$: $\mathcal{I}$ is then called a *model* of $\mathcal{F}$. A CNF formula $\mathcal{F}$ is said to be *satisfiable* if there exists an assignment under which $\mathcal{F}$ is satisfied ; $\mathcal{F}$ is *unsatisfiable* otherwise. Given a CNF formula $\mathcal{F}$ and a literal l, we write $\mathcal{F}_{|l} = \{c | c \in \mathcal{F}, \{l, \neg l\} \cap c = \emptyset\} \cup \{c \backslash \{\neg l\} | c \in \mathcal{F}, \neg l \in c \text{ and } l \notin c\}$. $\mathcal{F}_{|l}$ denotes the simplified formula obtained from $\mathcal{F}$ by removing all clauses $c \in \mathcal{F}$ such that $l \in c$ and $\neg l$ from clauses containing it. This simplification can be extended to a set of literals $\{l_1, \cdots l_k\}$; thereby $\mathcal{F}_{|\{l_1,\cdots,l_k\}}$ is the formula obtained from $\mathcal{F}$ by successively applying the previous simplification rule on $l_1, l_2, \cdots$ and l_k i.e. $\mathcal{F}_{|\{l_1,\cdots,l_k\}} = (...(\mathcal{F}_{|l_1})_{|l_2}...)_{|l_k}$. *Unit propagation* is the application of the rule $\mathcal{F}_{|x}$ for each unit clause $\{x\} \in \mathcal{F}$ until a clause in $\mathcal{F}$ is falsified or $\mathcal{F}$ does not contain unit clauses anymore.

The Boolean Satisfiability Problem (SAT) consists of deciding whether a given propositional formula $\mathcal{F}$ is satisfiable or not; in other words, SAT consists of determining if there exists a truth assignment to variables of $\mathcal{F}$ which can make it evaluate to true. This problem is the canonical NP-complete problem [1], thus all known algorithms for SAT are exponential in the worst case. A software program built to solve SAT is called a SAT solver. SAT solvers can be divided into two categories: complete and incomplete. Incomplete solvers unlike complete ones do not guarantee that a given formula will be solved even with unlimited resources (time and memory). In this paper, we only focus on complete SAT solvers and their parallelization. Since many parallel SAT solvers are built on top of sequential ones, it is important to understand sequential algorithms for SAT in order to make their parallelization more efficient.

2.2 Sequential SAT solving

Most sequential SAT solvers are based on the CDCL (Conflict-Driven Clause Learning) procedure which relies on various features heavily improved over the years. These features include fast unit propagation using watched literals [7, 13], learning mechanisms [4], deterministic and randomized restart strategies [6, 10, 19], effective constraint database management and smart static and dynamic branching heuristics [2, 9]. We briefly describe some of these features next.

Branching heuristics: they are used to choose the next variable on which the solver will branch. They can significantly impact the efficiency of the solver since different selection order can lead to completely different performance. This feature has been the subject of special attention and many branching heuristics have been proposed in the literature. Some of them are said to be static because they do not take into account dynamic information generated during the search process: examples of such heuristics are MOMS [9] and JW [2]. The other ones are dynamic since they make use of information collected during the search process to determine the next branching variable. One of the most used is the VSIDS heuristic introduced by [12] in the solver zChaff: it gives more priority to variables involved in recent conflicts analysis when picking a decision variable. To do that, VSIDS maintains activity scores for each variable and always selects the most active variable (i.e. the variable with the highest score) for branching. Whenever a variable appears in conflict analysis process, its activity is increased by a certain amount. In order to favor recent variables, activity scores of all variables are reduced periodically.

Watched literals: they were introduced in [7, 13] in order to perform unit propagation more quickly and consequently to detect conflicts more effectively. The principle of watched literals is to maintain for each clause a pair of two literals not falsified by the current interpretation called watched literals: initially, any pair of literals can be watched for each clause since all literals are unassigned. Whenever a literal is assigned, a clause is visited only if one of its two watched literals is falsified. In this case, a non-falsified literal is searched in order to replace the falsified one. For this, three situations can occur:

1. The second watched literals is satisfied: in this case, nothing needs to be done.
2. One non-watched satisfied or unassigned literal is found: this becomes the new watched literal.
3. No non-watched satisfied or unassigned literal is found: if the second watched literal is unassigned then the clause is unit under the current assignment and its unique literal is introduced into the propagation queue in order to be propagated later. If the second watched literal is falsified then there is a conflict since the clause is falsified.

One of the main advantages of this data structure is that the watched literals do not need to be updated when backtracking.

Clause learning: learning [4] is considered as one of the most important features in CDCL SAT solvers. The idea here is to examine each conflict, produce a learned clause representing its cause and store it in a learned clause database. This will help further to prune the search space and thereby, to be more efficient. The learning process makes use of a data structure called implication graph which records the partial assignment under construction made of the successive decision literals with their propagation. Whenever a conflict is encountered, a learned clause or no-good is generated thanks to a traversal of the implication graph using the resolution mechanism. This is done by generating resolvents step by step beginning with the conflict clause until obtaining an asserting clause (i.e. a clause containing exactly one literal of the conflicting decision level and that is falsified by the current interpretation). Each step consists of a resolution between the current clause and the reason clause of the negation of its most recently assigned literal of the conflicting decision level. This asserting clause helps redirect the backjumping level and is added to the learned clause database in order to prevent the same conflict in the future.

Restarts: they were first introduced by [6, 10] to cope with the heavy-tailed phenomenon. Restart consists of backtracking to the top level (level 0) in order to begin a new search. From one restart to another some information is kept in order to accelerate the new search. This information can be the set of learned clauses, the saved polarities, variable activities etc. There are several restart policies used in modern SAT solvers including luby restart which makes use of the luby series [3], arithmetic and geometric policies also based on series. Recently, a restart strategy based on LBD (Literal Bloc Distance) score [19] has been introduced and implemented in the solver Glucose [25].

2.3 Parallel SAT Solving

The two main approaches commonly used in parallel SAT solving are *search space splitting* and *portfolio*. Each of them has its strengths and its weaknesses. In this section, we aim to present those two approaches.

2.3.1 Search Space Splitting

In this approach, the search space is partitioned into several disjoint parts or branches which can be treated in parallel. The partition function used to split the search space takes as input a formula $\mathcal{F}$ and outputs a set $\mathcal{P} = \{\mathcal{F}_1, \mathcal{F}_2, \cdots, \mathcal{F}_n\}$ of sub-formulas such that $\mathcal{F}$ is satisfiable if there exists a satisfiable $\mathcal{F}_i \in \mathcal{P}$ and is unsatisfiable if every $\mathcal{F}_i \in \mathcal{P}$ is unsatisfiable. Due to the difficulty of predicting the time needed to complete a specific branch of the search space [26], the partitioning is usually done dynamically rather than statically. Therefore, parallel solvers based on search space splitting dynamically partition the search space, assigning available work to the available threads during runtime. The splitting of the search space is usually done by means of the so-called guiding path. This concept of guiding path was initially introduced by [5] and has been much used in parallel SAT solvers. It describes the current state of the search process by recording the list of variables to which the solver gave a value up until the current point of execution. Guiding paths are used to distribute work among threads during the search process. Since some guiding paths can be easier to solve than others, a workload balancing strategy such as dynamic work stealing [16] is used to supply tasks to idle threads during execution. Thus, when a thread becomes idle, it can request new guiding paths from another thread or from the master thread depending on the chosen collaboration scheme. The

search process is stopped when a model is found by one thread or when all guiding paths are solved. The main drawback here mentioned in [22] is the load balancing issue since it is hard to predict the time needed to complete a specific branch of the search tree and therefore difficult to find a partition that balances work among threads. In addition, using dynamic load balancing in the context of SAT can bring further issues such as the *Ping-Pong phenomenon* [11] which occurs when division of the search space using a variable repeatedly provides two subspaces with one that is very easy to solve. Hence, workers spend a huge amount of time on splitting operations and communications instead of actually solving the problem itself. Another issue is *useless division* [29] where the resulting sub-formulas are identical. These issues are the source of an important overhead in parallel SAT solvers based on search space splitting. However, through the splitting of the search space, good speedup can be reached more frequently on satisfiable formulas.

2.3.2 Portfolio

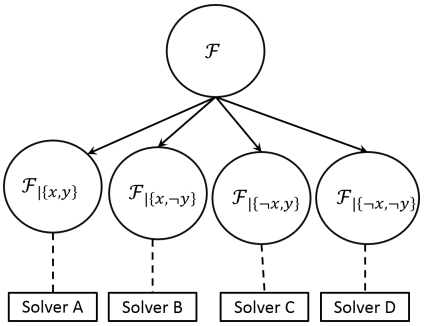
The portfolio approach exploits the complementarity between different sequential CDCL strategies that compete and cooperate on the same formula. To be efficient, a good crafting of the solver is required in order to perform the search in the best possible way. Portfolio solvers generally run different incarnations (also referred to as threads or solvers) of the same sequential solver on the same instance: the rationale is the high sensibility to parameter tuning which constitutes the main weakness of modern solvers [17]. For instance, a small change of parameters related to the restart strategy, the learned clauses database cleaning strategy or the branching heuristic can lead to a solver with completely different performances. Threads of the portfolio then use different parameters tuning that lead to complementary strategies in order to cover the search space in the best possible way. In order to improve the performance of the system beyond the performances of its individual threads, information sharing has been introduced in portfolio solvers. This information includes learned clauses, variable activities, equivalent variables etc. In consideration of the fact that there can be a large quantity of information to share between solvers of the portfolio, a selection criterion is used in order to limit the amount of information that a thread can send to or receive from the others. Therefore, only relevant information is exchanged: for instance, in some portfolio solvers, learned clauses with a small size or LBD (Literal Block Distance) [19] are considered relevant and hence, are shared amongst the solvers of the portfolio.

Portfolio has the advantage that it does not need load balancing and is simple to implement. However, a real challenge with portfolio approach is the difficulty to guarantee diversification of the search through algorithms that complement each other and therefore difficult to ensure scalability [26]. When performing diversification, it is important to intensify the search as well since excessive diversification can decrease performance. Thus, it is crucial to have a good control on the diversification and the intensification processes during the search.

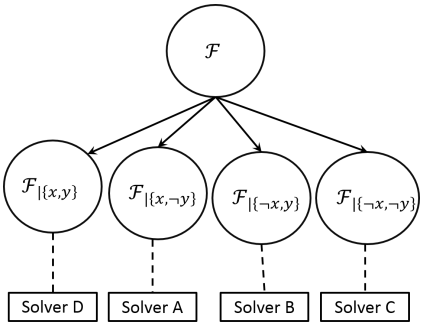
III OUR PARALLEL HYBRIDIZATION SCHEME

Decomposition in search space splitting is beneficial since it can help achieve better speedup while competition in portfolio with different search strategies can help explore the search space in different and complementary manners without the need of load balancing. It is then natural to think of a hybrid approach that can inherit those characteristics in order to perform better. We present in this section a new hybridization scheme for parallel SAT solving. The principle of our approach is described as follows:

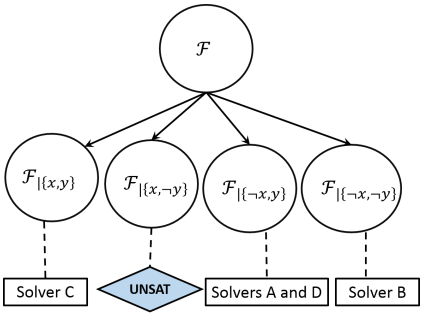
We start with the decomposition of the search space into multiple disjoint parts as in search space splitting approach (Fig. 1(a)). This decomposition can be carried out by any partition function and also, can be performed in parallel for more efficiency. At this level, the better the partition function is, the better the resulting algorithm. After this partitioning, solvers of the portfolio are placed in a regular manner over the different parts (Fig. 1(a)): in this way, the chances to quickly discover a solution on satisfiable benchmarks are increased. Each solver of the portfolio has its own strategy and migrates (or jumps) through subspaces (also referred to as subproblems, sub-formulas, parts or guiding paths) in a round robin fashion, looking for a solution (Fig. 1(b)). This migration is performed even if the current guiding path is not yet solved and is directed by a heuristic that helps threads escape from subspaces that seem not interesting. The threads could however branch to this guiding path later and with all additional information gathered during the search elsewhere, this subspace could be interesting again. Notice that we use here the *interestingness* but not the *hardness* since a subspace can seem difficult to solve according to a particular thread but very easy to solve by another one with a different strategy or by the same thread at some point in the future. The difference is that when a subspace seems difficult to solve according to a single thread or a subset of threads, then it is *uninteresting*. But when it seems difficult according to every thread after several attempts then it is considered *hard*. At this level the interestingness of the subspace can be expressed according to the number of conflicts achieved within it, the average learned clause sizes or LBD scores [19, 25] in this subspace, the evolution of the search process or any other measure. Whenever a solver encounters an unsatisfiable sub-formula, it marks it to prevent other solvers from branching to it again (Fig. 1(c)). When one solver finds a solution (i.e. a model) or when all sub-formulas are unsatisfiable, the search process is stopped (Fig. 1(f)). If it happens at some point of execution that only a single subspace remains to be explored (Fig. 1(d)), the solving process is temporarily stopped in order to repartition the remaining subspace (Fig. 1(e)) which is considered as the hardest one among the initial parts. The rationale is that when a single unsolved subspace remains, it means that several threads with different and complementary strategies and sometimes with multiple attempts have tried to solve it without success. This subspace is then not only considered as uninteresting but declared as hard or difficult and is therefore split again. Relevant learned clauses are still exchanged as in classical parallel solvers in order to improve the efficiency of the system. In addition, threads can perform several restarts on the same sub-formula: this can be useful since it helps achieve the same objectives as the standard restart strategy in CDCL SAT solvers but within a specific subspace.



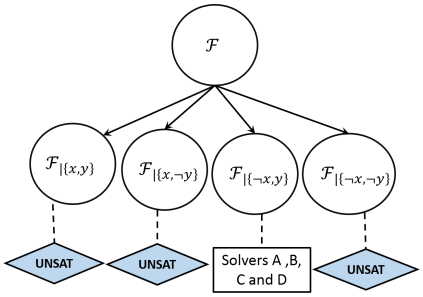
((a)) Initially, solvers of the portfolio are regularly distributed over sub-formulas



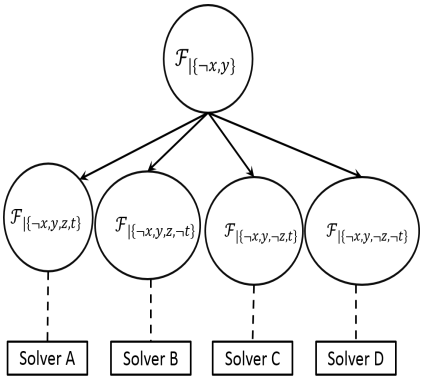
((b)) Solvers migrate in a round robin fashion from one sub-formula to another



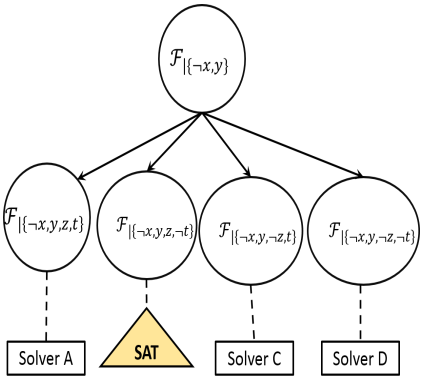
((c)) Unsatisfiable sub-formulas are marked in order to prevent other solvers from branching to it again



((d)) When it remains a single sub-formula, it is split again



((e)) Partitioning



((f)) The search is stopped if a solver finds a model or if all sub-formulas are unsatisfiable

Figure 1: Schematic representation of the different steps of our approach

With this hybridization scheme, we can benefit from the strengths of both parallel approaches while eliminating some of their individual weaknesses. At first, there is no need to introduce a workload balancing mechanism as in search space splitting since at no time in the solving process, a solver becomes idle. Parallel solvers based on our scheme are likely to reach more often a super-linear speedup through the splitting of the search space, and the use of a portfolio with multiple search strategies helps explore subspaces with different and complementary methods which therefore increases robustness. Furthermore, the use of a migration heuristic helps threads escape from uninteresting subspaces and consequently directs them toward subspaces

that are likely to be rapidly solved.

Useless splitting is no longer dramatic since even if it happens that threads work on identical sub-formulas, the various heuristics that they use help them explore it differently. The *ping pong phenomenon* is avoided here because each thread does not just work on a single part of the search space but instead, it works on the entire set of parts and no thread is stopped during the search in order to split its work. Furthermore, our approach is easier to implement compared to the search space splitting approach which requires dynamic work stealing.

Unlike classical portfolio, the diversification is well controlled through the splitting of the original search space into multiple disjoint subspaces. Thus, changing sub-formula can help improve diversification that is also enhanced by the use of many search heuristics and the initial placement of the threads over subspaces. As regards the intensification, a migration heuristic is used to control the amount of time a thread spends in a particular subspace.

In the literature, it is not clear how to characterize a hard subspace. The number of conflicts, the average LBD scores and the average backjumping levels are some measures commonly used to determine the hardness of a subspace. In the parallel context, these measures are sometimes taken according to a single thread. However, a subspace can seem difficult to a thread with its strategy while being very easy to solve by another thread with a different strategy. Moreover, even with a single thread, a subspace might seem difficult at the present moment but becomes very easy to solve later with additional information learned during the search performed elsewhere. In contrast, we consider a subspace hard when several threads having different search heuristics strengthened by information sharing have attempted to solve it without success. It is the case when during our proposed approach it remains a single sub-formula while the others have been proved unsatisfiable. Unlike some hybridization techniques which use a portfolio to solve difficult branches of the search space, once a sub-formula is found hard, then it is split again. The rationale is that when a task is difficult or large, it would be more natural to split it into small parts before solving rather than giving the whole task to each of the available workers.

Algorithm 1 describes our hybrid approach. In this algorithm, function *partition* is used to split the search space of the given formula into multiple disjoint parts: it returns an array of guiding paths representing the different search spaces. Function *solve* solves the current subformula until reaching the given limit expressed in terms of number of conflicts. It can return three possible values: *undefined*, *false* or *true*. When *undefined* is returned, it means that the current subformula has not been solved after *nbConflicts* conflicts; *false* means that the current subformula is unsatisfiable while *true* indicates that a model has been found. In this latter case, the search process is stopped and the solution is returned. *mark* is used to mark unsatisfiable subformulas in order to prevent other threads from branching to it again. *solved* checks whether the problem has been solved by another thread; if so, the search process is stopped. *setSolved* is used to inform other threads that the formula has been solved. *changeSubFormula* returns a Boolean value, indicating whether it is time to jump to another subformula or not. *getNextSubformula* searches and returns the next unsolved subformula if any or *undef* if no unsolved subformula was found. This function is responsible for choosing the migration policy: in our implementation, we chose to move them in a round robin fashion using a heuristic to decide the moment a thread must leave its current subspace. *waitForOtherThreads* places a synchronization barrier

so that all threads reach this point before the process continues.

Algorithm 1: Hybrid Parallel Algorithm for SAT

Input: A CNF formula $\mathcal{F}$
Result: SAT or UNSAT

```
1  begin
2      parts := partition(F);
3      waitForOtherThreads();
4      currentSubFormula := parts[threadId × parts.size()/nbThreads];
5      while true do
6          if solved() then return;
7          status := solve(currentSubFormula, nbConflicts);
8          if status = false then mark(currentSubFormula);
9          if status = true then
10             setSolved(true);
11             return SAT
12         end
13         if changeSubFormula() then
14             previousSubFormula := currentSubFormula;
15             currentSubFormula := getNextSubFormula();
16             if currentSubFormula = undef then
17                 setSolved(true);
18                 return UNSAT
19             end
20             if currentSubFormula = previousSubFormula then
21                 waitForOtherThreads();
22                 parts := partition(currentSubFormula);
23                 waitForOtherThreads();
24                 currentSubFormula := parts[threadId × parts.size()/nbThreads]
25             end
26         end
27     end
28 end
```

IV IMPLEMENTATION DETAILS AND EXPERIMENTS

We implemented our approach on top of the solver PENELOPE [28] (the 2014 SAT Competition version), a parallel portfolio SAT solver which is in turn built on top of MANYSAT [17] and MINISAT [14]. We gave to this modified version of PENELOPE the name PENELOPEDPRFOLIO. PENELOPE was chosen because of its good performance in previous SAT competitions and additionally because it is built on top of the famous SAT solvers MINISAT and MANYSAT that are well documented and easy to modify. Note that we only empirically compared our solvers with the base solver on which they are built but not other parallel hybrid SAT solvers or parallel solvers based on search space splitting. The reason is that most of these solvers are not easily available online or they are implemented for special environments using non-standard middlewares. Nonetheless, in Section V we compared our approach to others based on how they work.

For the partitioning heuristic, we used a weak portfolio [22] to choose 3 partition variables. The principle of weak portfolio is to run a first stage of portfolio for a small amount of time

usually expressed in term of number of conflicts. After that, variables that are the most active according to all threads of the portfolio are chosen to split the search tree. To do so, in each thread, variables are ranked in descending order according to their activities. Afterward, each variable is given a score which corresponds to the sum of its ranks in each thread. Finally, the variables with the lowest score (which correspond to the most active ones) are chosen for partitioning. Weak portfolio has the advantage that easy formulas can be solved without any splitting of the search space.

We also used assumptions [14] to indicate the guiding path each thread must branch on: in this manner, learned clauses could be shared amongst threads without restriction and unsatisfiable instances could be sometimes solved by a single thread when a top-level (level 0) conflict has been found i.e. without proving the unsatisfiability of all the parts of the whole partition. It is worth mentioning that when it remains a single unsolved guiding path $G = \{l_1, \cdots l_k\}$, then its literals can be considered as units since the whole formula is satisfiable if and only if $\mathcal{F}_{|G}$ is satisfiable. To decide the moment at which a thread jumps from one part to another, we used the following heuristic based on LBD [19, 25]: every thread is forced to make at least 100 conflicts in a subspace it branches to before any jump; unless the corresponding subformula is earlier found unsatisfiable. This is used to prevent threads from jumping every time without performing a significant search in the subspaces they just branch to. Each thread jumps from its current subformula to the next one whenever the average LBD scores of all learned clauses generated since the branching on this subspace multiplied by a constant α ($0 < \alpha < 1$) called *jump factor* is greater than the global mean of the LBD scores computed since the launching of the thread. The rationale here is that if in one subspace, a solver is learning clauses with bad LBD scores, then it may not be an interesting subspace and therefore, jumping to another one can prevent it from getting stuck in it. More formally, if M is the current mean of the LBD scores since the entering in the current subspace and M_G is the mean since the launching of the thread, then this thread must jump to the next unsolved subspace if $M \times \alpha > M_G$. This heuristic is similar to the one used in LBD restart strategy [25] but does not need the use of a bounded queue. According to the value of the jump factor α, we differentiated three versions of PENELOPEDPRFOLIO: PENELOPEDPRFOLIO-0.6, PENELOPEDPRFOLIO-0.7 and PENELOPEDPRFOLIO-0.9 with respectively the *jump factor* set to 0.6, 0.7 and 0.9. Notice that a *jump factor* close to 1 indicates that the thread jumps more frequently.

All our experiments were conducted on the StarExec[1] [30] cluster infrastructure. Each node of this infrastructure has two 4-core (2.4GHz) Intel processors, but we only had the possibility to use one of them. This means that we could only launch 4 threads in parallel and that is why all the solvers we used (PENELOPE included) were tuned to use 4 threads. We also used deterministic mode to ensure reproducibility.

Experiments were carried out on the 100 parallel track benchmarks of the SAT-Race 2015[2] and the 300 application benchmarks of the SAT Competition 2016[3]. Notice that the 100 benchmarks (out of the 300 benchmarks used in the SAT-Race 2015) of the SAT-Race were selected by the organizers based on their hardness using a measure presented in the competition page. Solvers were used without any preprocessing step. Each instance was given a wall clock time limit of 1800 seconds and a memory limit of 24 GB.

[1] https://www.starexec.org/
[2] https://baldur.iti.kit.edu/sat-race-2015/
[3] http://www.satcompetition.org/

Tables 1 and 2 summarize our results. In these tables, the number of solved instances is indicated for both satisfiable and unsatisfiable benchmarks and the total run time used to solve these instances is specified in brackets.

Solvers	#SAT (time)	#UNSAT (time)	Total
PENELOPE (SC 2014)	56 (11,931 s)	**64 (18,456 s)**	120
PENELOPEDPRFOLIO-0.6	**61 (16,802 s)**	61 (18,998 s)	**122**
PENELOPEDPRFOLIO-0.7	61 (17,216 s)	56 (11,713 s)	117
PENELOPEDPRFOLIO-0.9	56 (11,919 s)	54 (11,499 s)	110

Table 1: Experiment results on the 300 application benchmarks of the SAT competition 2016

Solvers	#SAT (time)	#UNSAT (time)	Total
PENELOPE (SC 2014)	26 (17,639 s)	7 (5,447 s)	33
PENELOPEDPRFOLIO-0.6	27 (11,826 s)	**7 (4,058 s)**	34
PENELOPEDPRFOLIO-0.7	27 (12,161 s)	7 (4,221 s)	34
PENELOPEDPRFOLIO-0.9	**32 (22,504 s)**	6 (3,393 s)	**38**

Table 2: Experiment results on the 100 hardest benchmarks of the parallel track of the SAT-Race 2015

In Table 1 we can notice that PENELOPEDPRFOLIO-0.6 solved 2 instances more than PENELOPE. But, the main improvement is on satisfiable instances where each of the solvers PENELOPEDPRFOLIO-0.6 and PENELOPEDPRFOLIO-0.7 solved 5 more satisfiable instances than PENELOPE. Fig. 2 shows cactus plot respectively on all benchmarks and on the satisfiable benchmarks. We can notice that our hybrid approach slightly improves the pure portfolio approach especially on satisfiable instances. Table 1 also suggests that the splitting of the search space can have negative impact on unsatisfiable instances since PENELOPE solved more unsatisfiable instances than PENELOPEDPRFOLIO and sometimes with smaller run times.

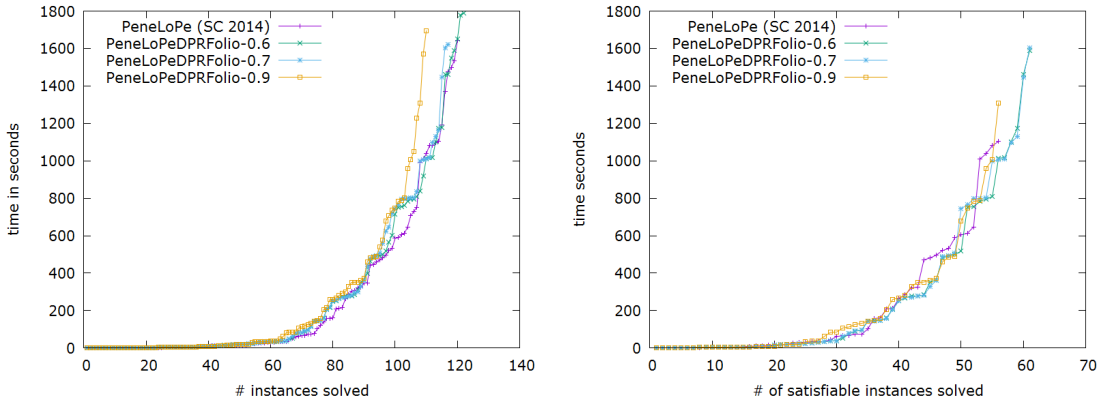


Figure 2: Cactus plot on the 300 application benchmarks of the SAT Competition 2016: on the left, the cactus plot on all (satisfiable and unsatisfiable) instances. On the right, we have the cactus plot on satisfiable instances only

Table 2 reports results on the hardest benchmarks of the SAT-Race 2015 and clearly indicates that our hybrid approach outperforms the original solver PENELOPE especially on satisfiable

instances where our solvers solved respectively 1 and 5 satisfiable instances more than PENELOPE. Furthermore, we can notice that the main improvement is not the additional number of solved instances but the total run time used to solve these instances. We have for instance, PENELOPEDPRFOLIO-0.6 which solved 27 satisfiable instances in 11,826 seconds while PENELOPE solved 26 instances in 17,639 seconds; so, the time used by PENELOPE to solve 26 instances is far greater than the time needed by PENELOPEDPRFOLIO-0.6 to solve 27 instances. In Fig. 3 we have the cactus plots on the total solved benchmarks and on the total satisfiable solved instances. These plots clearly show that PENELOPEDPRFOLIO outperforms PENELOPE and that the performance is mainly gained on satisfiable instances. This can be explained by the introduction of search space splitting in our approach which helps improve speedup on satisfiable instances.

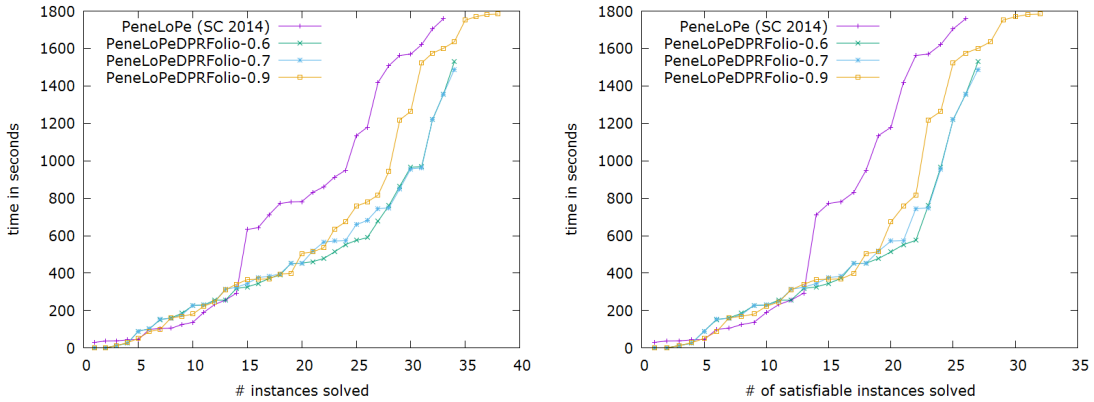


Figure 3: Cactus plot on the 100 hardest benchmarks of the SAT-Race 2015: on the left, the cactus plot on all (satisfiable and unsatisfiable) instances. On the right, we have the cactus plot on satisfiable instances only

These results lead us to the conclusion that our solvers perform well on hard satisfiable instances.

Despite this improvement gained on satisfiable instances, it is important to point out that the introduction of the search space splitting brought some slowness on the resolution of unsatisfiable instances. Although PENELOPEDPRFOLIO solved 7 unsatisfiable instances with less run time than PENELOPE on the SAT-Race 2015 benchmarks, PENELOPEDPRFOLIO did not perform well on unsatisfiable benchmarks of the SAT competition 2016. For instance, PENELOPEDPRFOLIO-0.6 solved 61 instances in 18,998 seconds which is greater than the time needed by PENELOPE to solve 64 instances. This can be explained since sometimes the splitting process can produce some parts that are almost as difficult as the original problem (known in the literature as *bogus split* [23]) and that the search space splitting sometimes needs to solve all subformulas before asserting the unsatisfiability of the whole formula.

V RELATED WORK AND DISCUSSION

Many hybrid approaches for parallel SAT solving have been proposed over the years.

BLOCHINGER [15] proposed to use an adaptive competition by starting with a search space splitting strategy and switching into a portfolio approach when a particular hard region of the search space is encountered. It starts with an exploratory decomposition and adaptively induces competition parallelism when the solving process of a particular subproblem does not make

sufficient progress. A transition heuristic is used in order to steer the transition from decomposition to competition parallelism of solving processes of individual subproblems. When the transition is initiated, the respective subproblem is treated using different heuristics and further decomposition of the latter is disabled. As we can see, this approach needs to balance workload between threads during the search space splitting phase; that is why the author used dynamic work stealing for that end. In addition, the hardness of a subspace is only determined by a single thread with a single configuration. In contrast to this latter remark, in our approach, the hardness of a subspace is determined by all the threads participating in the portfolio and is set as such when several threads with various configurations, strengthened by shared information have unsuccessfully attempted to solve it.

OHMURA *et al.* [21] in their solver C-SAT tried to take advantage of a high number of machines by combining search space splitting with a portfolio approach. They used a three-layered master-slave architecture including grandmasters, masters and workers. The grandmaster is only responsible for lemma exchange with the detection and removal of redundant clauses. Master threads as far as they are concerned partition the search tree using guiding paths so as to balance work among workers. Each worker uses a different heuristic and random seed to perform the search on the guiding path received from its master. Here again, dynamic workload balancing is necessary to prevent idleness of workers. Furthermore, since a worker can only abandon a subspace when the latter is solved (satisfiable or unsatisfiable), then a worker can get stuck into a subspace that seems to be very difficult for it while there are some other subspaces that it can solve very quickly. To overcome this limitation, we allowed threads to temporarily leave one part in favor of another even if their current subformula is not yet solved. These threads could however come back later to the abandoned subspace and the additional information gathered during search may help them solve it more efficiently.

NISHANT *et al.* [27] proposed to use a search space splitting at the high level to divide the search space into multiple disjoint parts and assign each part to a portfolio of solvers. They do not use any kind of workload balancing in their methods; hence processes that are assigned easy guiding paths rapidly become idle.

MARTINS *et al.* [22] proposed to begin with an initial stage of search space splitting, switching to a portfolio approach when load balancing becomes an issue or when a cutoff is reached. After this switch, the solver does the remaining work in a portfolio mode. The motivation of the authors is to use search space splitting when this approach is more efficient and to change to a portfolio approach when difficulties arise. The transition between the two modes is heuristically done. Here once again the dynamic load balancing is necessary before the transition. In addition, the transition between search space splitting and portfolio is initiated according to the point of view of a single method since before the transition all the threads use the same heuristic.

In contrast to previous approaches, instead of assigning a single task or a subset of tasks to a particular thread, we assign all the tasks to each of the available threads and offer them the ability to temporarily leave unsolved tasks to the detriment of others. Hence, when a subspace seems to be uninteresting, a thread can momentarily leave it and come back to it later in the hope of solving the latter more efficiently with all additional information learned during the search on other subspaces such as learned clauses, variable activities etc. As they do not begin at the same part or branch, some of them may luckily find a solution very quickly. Since each thread works on the whole partition, there is no need to implement work stealing to balance the load during the search and therefore, dedicated threads such as masters or grandmasters used to perform work

sharing and clause exchange among workers are no longer necessary. Furthermore, competition parallelism in our approach is achieved in two ways [18] over subformulas: the parallel portfolio where several threads are solving the same subproblem at the same time and the sequential portfolio where threads successively treat subformulas one after the other.

VI CONCLUSION AND FUTURE WORK

In this paper, we have presented a new parallel hybridization scheme for SAT. Our approach divides the search space into disjoint parts and then, places the solvers of the portfolio over these parts in a regular manner and lets them migrate from one part to another even if the current part is not yet solved. It uses heuristics for the choice of partition variables and the migration moment. Our approach does not need any workload balancing mechanism and can achieve good speedup on hard satisfiable instances. We integrated it in the solver *PeneLoPe* and performed some experiments and comparisons. The results showed that our hybridization scheme actually help improve the performance of the solver *PeneLoPe* especially on hard satisfiable instances.

Results suggest that the jump factor α may have a significant impact on the performance of the solver. So one further research direction is to investigate the real impact of this factor in the search process.

It might also be interesting to study how to limit the negative impact of search space splitting on unsatisfiable instances. For example we could introduce independent threads in the portfolio that are not required to branch on any guiding path. We could also allow threads to have some steps where they run on the whole formula without branching on any guiding path.

REFERENCES

Publications

[1] S. A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pages 151–158.

[2] R. G. Jeroslow and J. Wang. “Solving propositional satisfiability problems”. In: *Annals of mathematics and Artificial Intelligence* 1.1-4 (1990), pages 167–187.

[3] M. Luby, A. Sinclair, and D. Zuckerman. “Optimal speedup of Las Vegas algorithms”. In: *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*. IEEE. 1993, pages 128–133.

[4] J. Marques-Silva and K. Sakallah. *Grasp-a new search algorithm for satisfiability. ICCAD*. 1996.

[5] H. Zhang, M. P. Bonacina, and J. Hsiang. “PSATO: a distributed propositional prover and its application to quasigroup problems”. In: *Journal of Symbolic Computation* 21.4 (1996), pages 543–560.

[6] C. P. Gomes, B. Selman, and N. Crato. “Heavy-tailed distributions in combinatorial search”. In: *Principles and Practice of Constraint Programming-CP97*. Springer, 1997, pages 121–135.

[7] H. Zhang. “SATO: An efficient prepositional prover”. In: *Automated Deduction–CADE-14*. Springer, 1997, pages 272–275.

[8] C. P. Gomes, B. Selman, H. Kautz, et al. “Boosting combinatorial search through randomization”. In: *AAAI/IAAI* 98 (1998), pages 431–437.

[9] J. Marques-Silva. “The impact of branching heuristics in propositional satisfiability algorithms”. In: *Portuguese Conference on Artificial Intelligence*. Springer. 1999, pages 62–74.

[10] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. “Heavy-tailed phenomena in satisfiability and constraint satisfaction problems”. In: *Journal of automated reasoning* 24.1-2 (2000), pages 67–100.

[11] B. Jurkowiak, C. M. Li, and G. Utard. “Parallelizing Satz using dynamic workload balancing”. In: *Electronic Notes in Discrete Mathematics* 9 (2001), pages 174–189.

[12] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. “Chaff: Engineering an efficient SAT solver”. In: *Proceedings of the 38th annual Design Automation Conference*. ACM. 2001, pages 530–535.

[13] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. “Efficient conflict driven learning in a boolean satisfiability solver”. In: *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press. 2001, pages 279–285.

[14] N. Eén and N. Sörensson. “An extensible SAT-solver”. In: *Theory and applications of satisfiability testing*. Springer. 2003, pages 502–518.

[15] W. Blochinger. “Towards Robustness in Parallel SAT Solving.” In: *PARCO*. 2005, pages 301–308.

[16] B. Jurkowiak, C. M. Li, and G. Utard. “A parallelization scheme based on work stealing for a class of SAT solvers”. In: *Journal of Automated Reasoning* 34.1 (2005), pages 73–101.

[17] Y. Hamadi, S. Jabbour, and L. Sais. “ManySAT: a parallel SAT solver”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 6 (2008), pages 245–262.

[18] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. “SATzilla: portfolio-based algorithm selection for SAT”. In: *Journal of artificial intelligence research* 32 (2008), pages 565–606.

[19] G. Audemard and L. Simon. “GLUCOSE: a solver that predicts learnt clauses quality”. In: *SAT Competition* (2009), pages 7–8.

[20] A. Biere, M. Heule, and H. van Maaren. *Handbook of satisfiability*. Volume 185. IOS press, 2009.

[21] K. Ohmura and K. Ueda. “c-SAT: A Parallel SAT Solver for Clusters”. In: *Theory and Applications of Satisfiability Testing-SAT 2009*. Springer, 2009, pages 524–537.

[22] R. Martins, V. Manquinho, and I. Lynce. “Improving search space splitting for parallel SAT solving”. In: *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*. Volume 1. IEEE. 2010, pages 336–343.

[23] S. Schulz and W. Blochinger. “Cooperate and compete! A hybrid solving strategy for task-parallel SAT solving on peer-to-peer desktop grids”. In: *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE. 2010, pages 314–323.

[24] S. Hölldobler, N. Manthey, P. S. Van Hau Nguyen, and J. Stecklina. “Modern parallel SAT-solvers”. In: *TR 2011–6* (2011).

[25] G. Audemard and L. Simon. “Refining restarts strategies for SAT and UNSAT”. In: *Principles and Practice of Constraint Programming*. Springer. 2012, pages 118–126.

[26] R. Martins, V. Manquinho, and I. Lynce. “An overview of parallel SAT solving”. In: *Constraints* 17.3 (2012), pages 304–347.

[27] N. Totla and A. Devarakonda. *Massive Parallelization of SAT Solvers [cs262a Project Report]*. 2013.

[28] G. Audemard, B. Hoessen, S. Jabbour, J.-M. Lagniez, and C. Piette. “PeneLoPe in SAT Competition 2014”. In: *SAT COMPETITION* (2014), page 58.

[29] G. Audemard, B. Hoessen, S. Jabbour, and C. Piette. “An effective distributed d&c approach for the satisfiability problem”. In: *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*. IEEE. 2014, pages 183–187.

[30] A. Stump, G. Sutcliffe, and C. Tinelli. “StarExec: a cross-community infrastructure for logic solving”. In: *International Joint Conference on Automated Reasoning*. Springer. 2014, pages 367–373.

[31] T. Menouer and S. Baarir. “Parallel Satisfiability Solver Based on Hybrid Partitioning Method”. In: *2017 25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2017, pages 54–60.

[32] T. Balyo and C. Sinz. “Parallel satisfiability”. In: *Handbook of Parallel Constraint Reasoning*. Springer, 2018, pages 3–29.

[33] L. Simon. “Reasoning with Propositional Logic: From SAT Solvers to Knowledge Compilation”. In: *A Guided Tour of Artificial Intelligence Research*. Springer, 2020, pages 115–152.